

LAZARUS Open Source version of, and almost compatible with: **DELPHI:**

At this website there are some downloads for Lazarus: <http://www.osalt.com/lazarus>

And locate the download link: http://sourceforge.net/project/showfiles.php?group_id=89339

and locate the Windows 32 bit version

YES	lazarus-0.9.26-fpc-2.2.2-win32.exe Mirror	58455268	i386
NO	lazarus-qt-0.9.26-fpc-2.2.2-win32.exe Mirror	58420736	i386

the version for Windows XP was about 58mb: [lazarus-0.9.26-fpc-2.2.2-win32.exe](#)

but do NOT download: [lazarus-qt-0.9.26-fpc-2.2.2-win32.exe](#)
because you will get very frustrated trying to locate: [qtcore4.dll](#)

documentation is available online at a url something like:-

http://wiki.lazarus.freepascal.org/Lazarus_Documentation#Lazarus_and_Pascal_Tutorials

The download is one single file, installs first time, and runs first time. The tutorial to get started is helpful, and located at:-

<http://www.lazarus.freepascal.org/>

http://wiki.lazarus.freepascal.org/Lazarus_Tutorial

WINDOWS VISTA win64 ~ ~ ~ STRIP and UPX **do not work** on win64, however WINZIP will still do a very good compression job. Also, win64 Lazarus code **does not work** on XP and win32. So stick with the win32 version unless you need win64 code. Sometimes **Lazarus waits** after RUN on Vista (32 bit version), click RUN, RESET DEBUGGER and it will work the next time.

DELPHI ~ ~ ~ This can be compared to DELPHI, which is 332 mb, and the pre-reqs another 234 mb. Delphi can be found at: <http://www.turboexplorer.com/delphi>

COMMENTS

LAZARUS is an Open Source program, based on PASCAL, and is somewhat compatible with Delphi.

One of the shortcomings of JAVA, and other object oriented languages is their type conversion issues. The graphical program for sundials highlights this problem, namely converting from floating point to integer requires an intermediate string conversion! Harking back to IBM's PL/I language, some lessons that the new language developers could learn emerge.

First: PL/I had as one of its values the concept that if a programmer could write something that made sense to him or her, then the PL/I compiler should also be able to make sense of it. All these newer languages or language adaptations are very weak on real world needs of commercial programmers, and seem to be more suited to those who delight in getting around complexities of a language. LAZARUS is no exception, and the documentation is designed for those who already know the system.

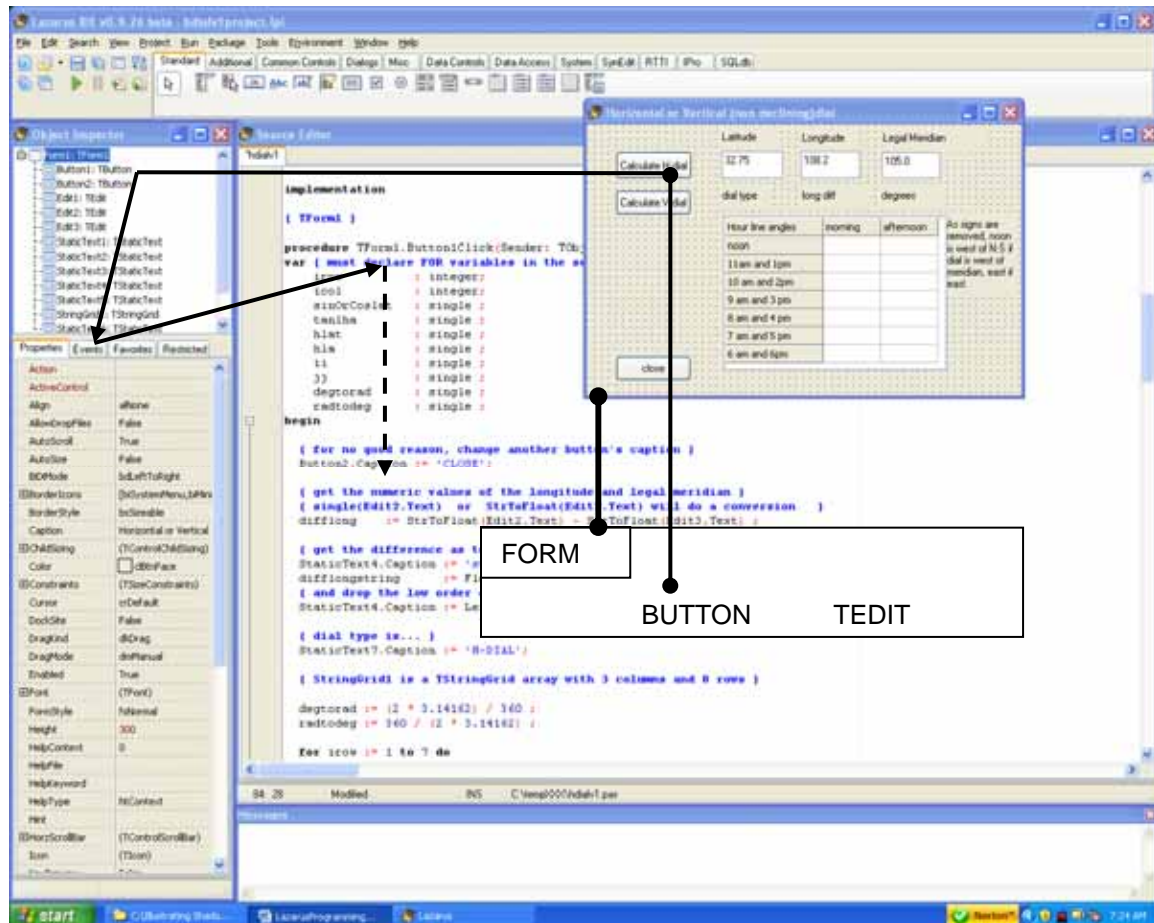
Second: PL/I had the ability for almost any data type to be converted implicitly so that a programmer could take in a string of characters that contained numbers and implicitly convert it to an integer, or floating point number, and vice versa.

A PROJECT'S HIERARCHY

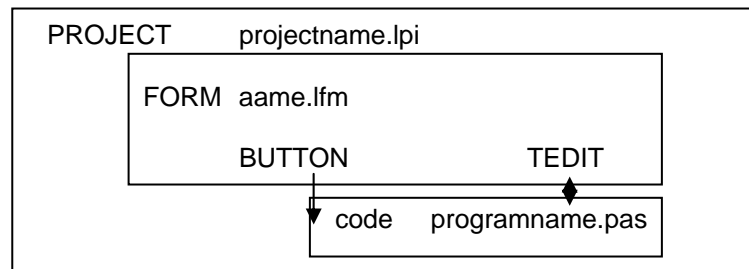
In LAZARUS, as in Microsoft's VISUAL languages, there is a hierarchy. In a conventional program there is always a main program, and it calls sub programs, that use functions as well as language structure.

Lazarus, based on PASCAL, at least does not confuse inherent language function with functions and classes. Some languages take classes to extreme and the blurring of those discrete boundaries causes programmers to fight the language in order to achieve the end goal.

There is a hierarchy in Lazarus similar to that of Visual Basic. The front end is the FORM.

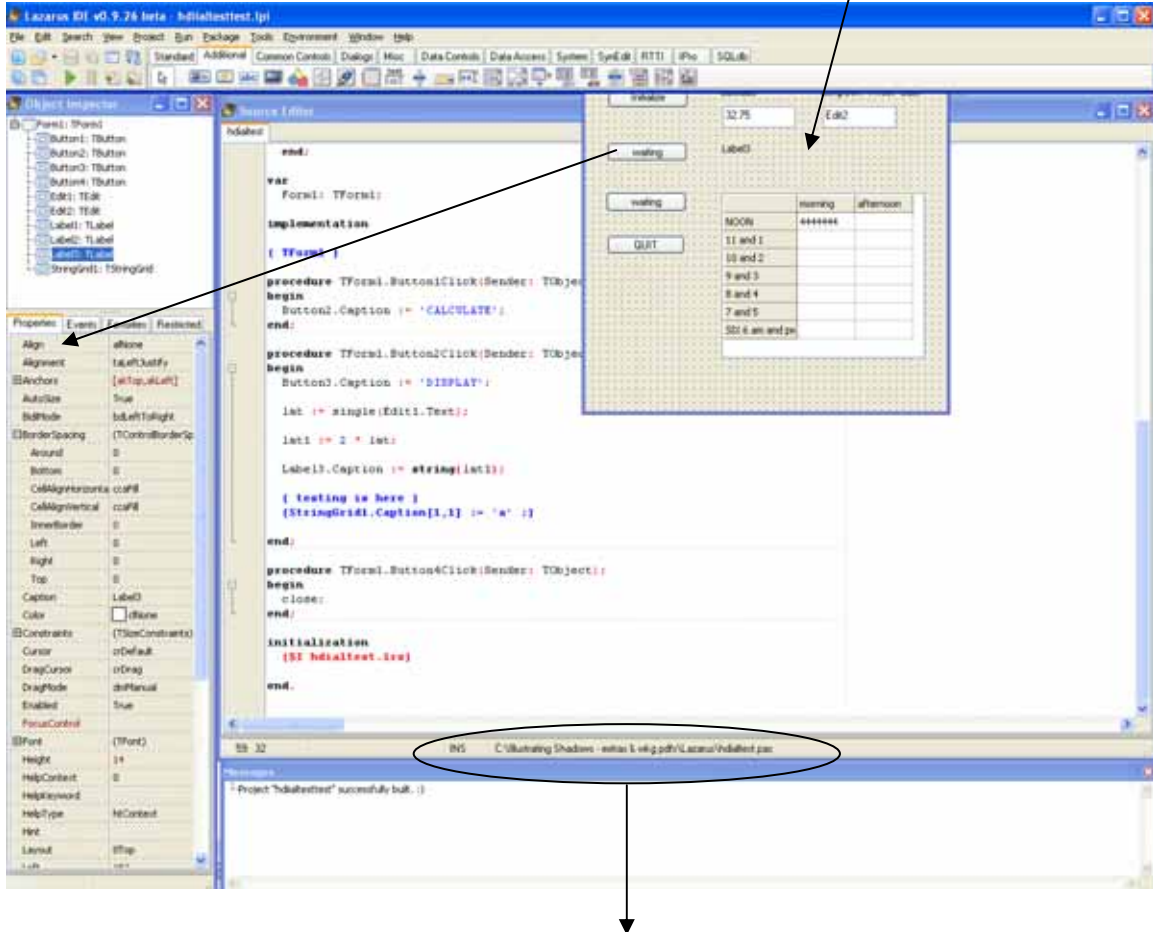


The FORM is what essentially starts up when the "project" is executed, and one or more buttons in the form trigger code. And that code can use variable data such as the TEDIT areas.



NOTES ON LAZARUS

Lazarus has an IDE and the forms window is simple to use to design a form that will drive the program. The content and activities for each button are intuitive.



Running a program that you built as an application may cause the debugger to crash. The debugger is how the IDE runs programs. However, locate the folders in which the programs are stored, bring up those folders and double click the program, and all will run, albeit without debugging. The folder containing the program can be found by SAVE AS, or by looking at the area circled in the about picture of the Lazarus IDE desktop.

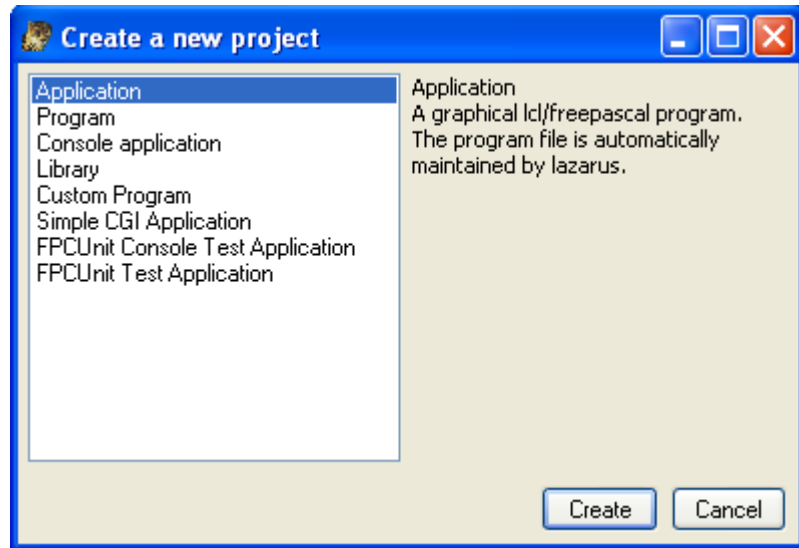
The cause of Lazarus problems in execution is simple, the designers and implementers did not manage folders with blanks in a name. So, if you wish to debug in Lazarus, you must save files in a folder or chain of folders whose names from the root folder up to the folder holding the project, have no names with blanks in them.

This issue of blanks in a name is not uncommon, so simply create folders with no blanks in their names for Open Source systems.

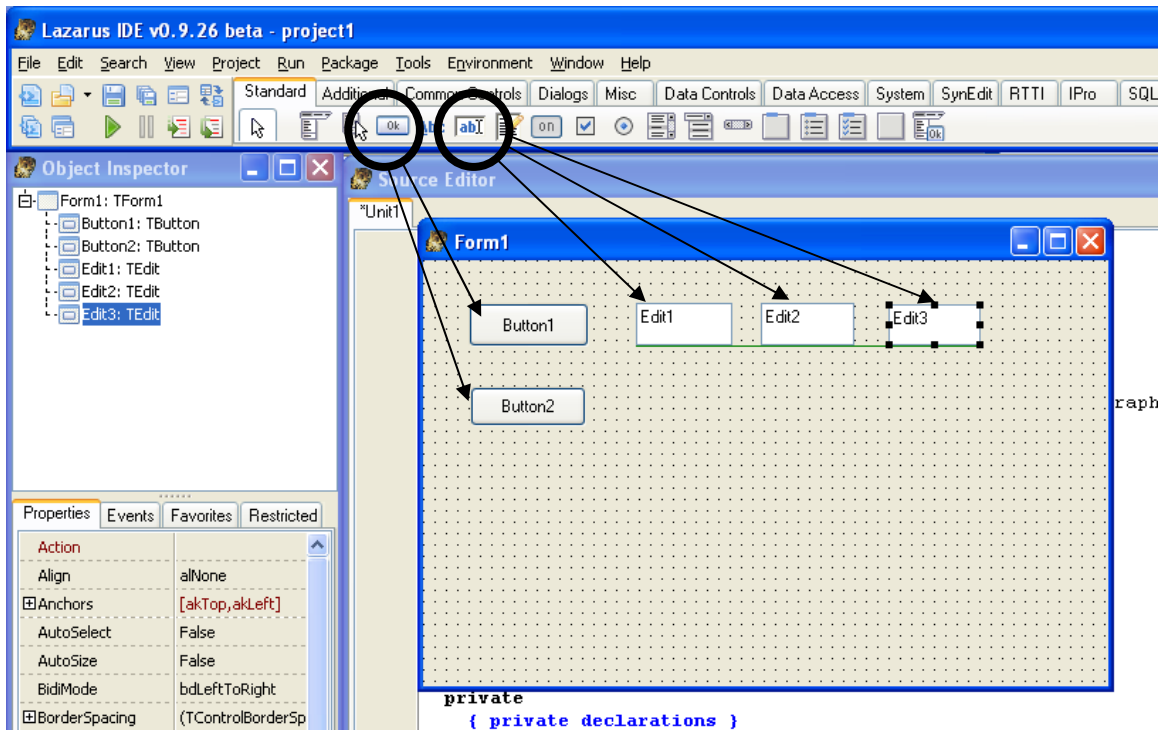
LAZARUS programming and systems ~ an open source Delphi equivalent

CREATING A PROGRAM ~ TABULAR HOUR LINE ANGLES ~ [hdialv1.pas](#)

Open up Lazarus, and select PROJECT and then NEW which brings up a small window.



Select APPLICATION and then click CREATE.

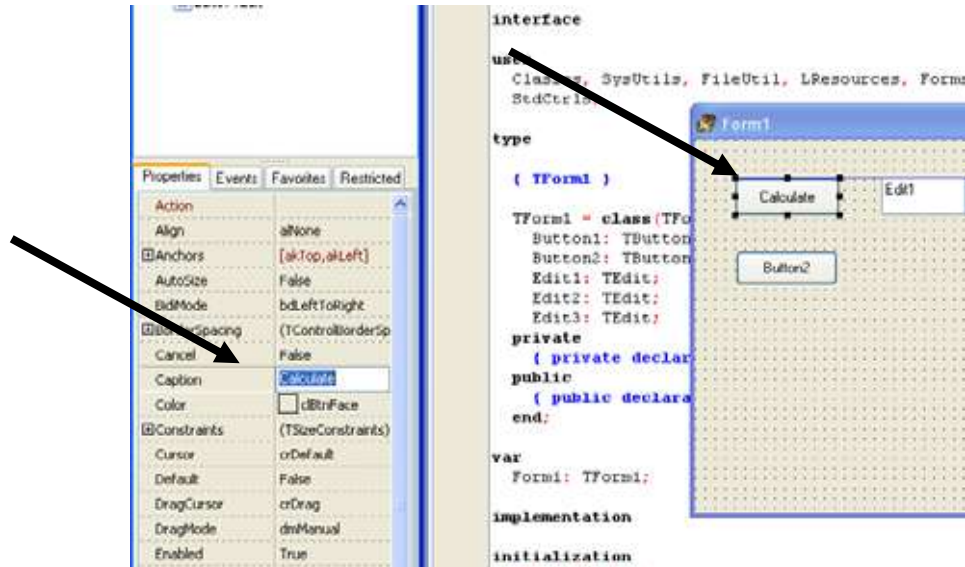


Lazarus will bring up a blank form. Select some options from the tool bar as shown and create a form.

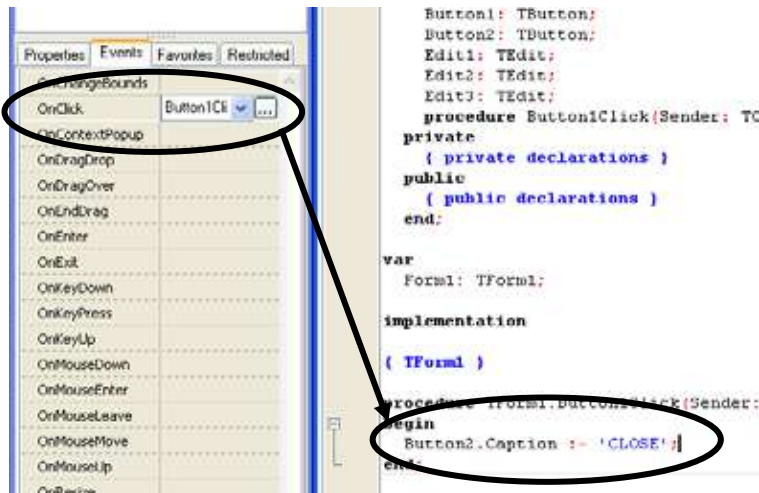
LAZARUS programming and systems ~ an open source Delphi equivalent

In doing this, the FORM will become what the end user sees when the application (program) is started.

Select a Button, and that button's details will appear off to the left of the IDE (integrated development environment).



Altering the button's caption as shown above, also changed the form as displayed. You may wish the "calculate" code to change the caption of the second button to "close". Simply select the EVENTS tab of the CALCULATE button, and click the "...", and you will be pointed to code in the program that will run when that button is clicked.



The code entered changes the second button's caption to "CLOSE" which doesn't do anything yet other than change the caption.

Then for the second button repeat the process but for the ONCLICK event, CLOSE the application.

The code would look something like the following:-

LAZARUS programming and systems ~ an open source Delphi equivalent

```
TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
private
    { private declarations }
public
    { public declarations }
end;

var
    Form1: TForm1;

implementation

{ TForm1 }

procedure TForm1.Button1Click(Sender: TObject);
begin
    Button2.Caption := 'CLOSE';
end;

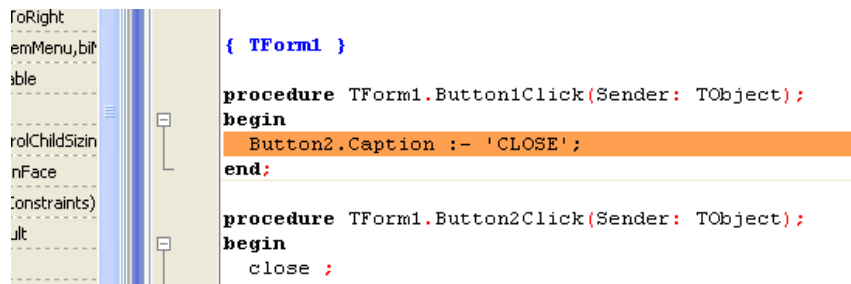
procedure TForm1.Button2Click(Sender: TObject);
begin
    close ;
end;
```

At this point the project should be saved and in a folder or chain of folders with no blanks in their names.

PROJECT, SAVE PROJECT AS and enter a project name such as: hdiav1.pas where Lazarus entered the suffix ".pas".

No sooner than you have done that, but it comes up with a second save panel but this time the suffix is ".lpi", and the name cannot be the same as the one you just used. The name chosen for this project was: hdiav1project.lpi and it was saved. Exit Lazarus and then start Lazarus again and make sure it finds everything.

The project was located automatically although you could open any project, and RUN then RUN or F9 used to start execution. The compiler hit an error and highlighted it in the IDE edit window.



The code typo was changed from `close` to `Close` and recompiled, and the program did as it was expected to do.

LAZARUS programming and systems ~ an open source Delphi equivalent

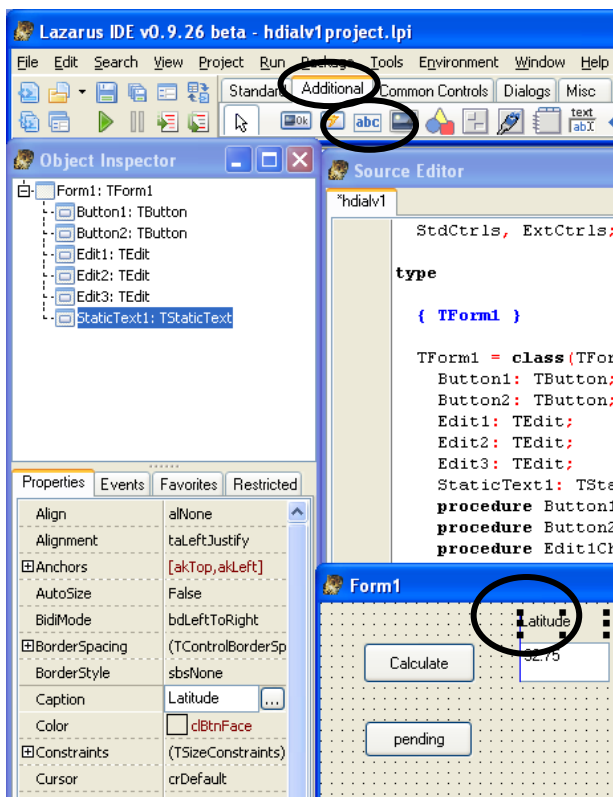
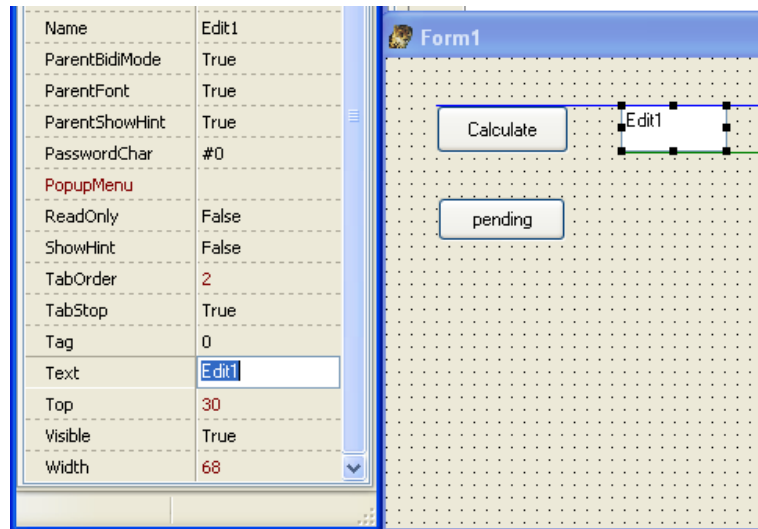
The program has managed an event when one button was clicked which changed the caption of a second button, and that second button has allowed the program to close. Any caption of any element on a form can be changed.

A couple of notes at this stage. First, folders in the Lazarus system cannot have blanks. Second, while you can enter the same name for the .PAS and the .LPI files being saved, and they will be saved, all sorts of problems happen when executed or when reloaded. This caution is made clear in the Lazarus documentation, and they mean it.

The time has come to add some more things on the FORM, which has vanished from the IDE. Click WINDOW and then FORM and it will reappear.

The first Edit element on the form is called Edit1 and it does not have a caption, instead it has a TEXT field whose default content is the element's name, namely "Edit1".

This Text area was changed to "32.75" being default data for the latitude. This did not change the elements name which is still "Edit1".

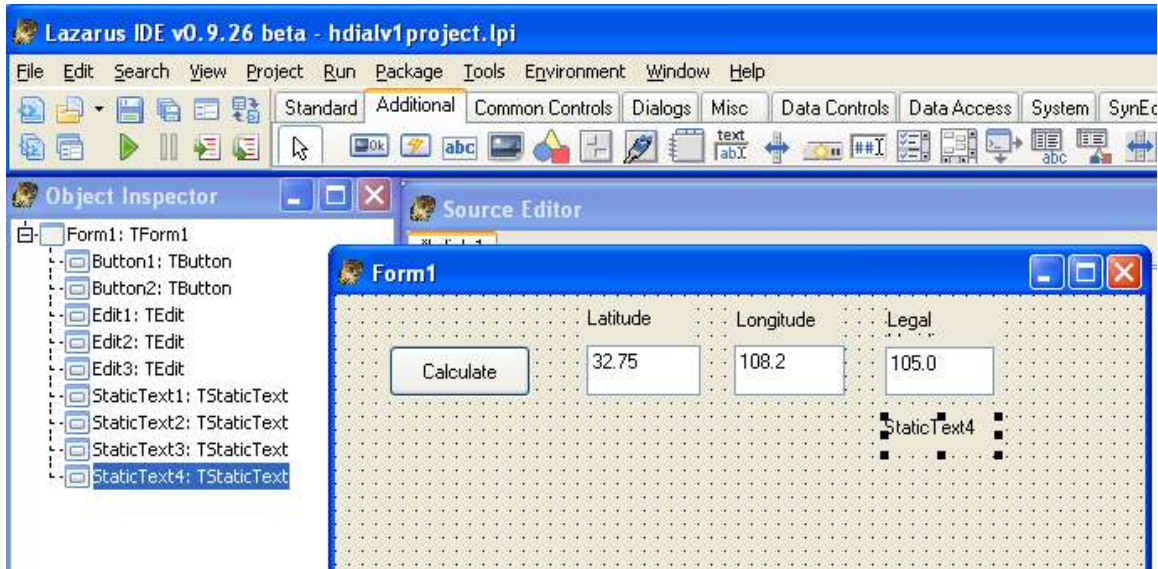


Click on ADDITIONAL on the tool bar for more FORM options, and select ABC which is for "static text". This allows you to place headings on the form.

The heading "LATITUDE" was chosen.

LAZARUS programming and systems ~ an open source Delphi equivalent

At this point the longitude and legal meridian were added. Now the longitude difference needs to be calculated.



The left panel shows the names of the elements in the form, and the form itself can be seen. Code must be added to the CALCULATE button to select the longitude and legal meridian, which are of course in TEXT form, convert them from text to numeric (as single precision in this case), and display the results back as TEXT.

A globally accessible variable of LONGDIFF was added and code for the conversion as well.

type

```
{ TForm1 }

TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    ...
    ...
    ...

private
    { private declarations }
public
    { public declarations }
    longitude      : single;
    legalmeridian  : single;
    diffalong      : single;
end;
```

This provides common variables accessible anywhere.

CONVERTING THE STRINGS INTO NUMERICAL VALES and subtracts them for future use.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  { for no good reason, change another button's caption }
  Button2.Caption := 'CLOSE';
  { get the numeric values of the longitude and legal meridian }
  difflong := single(Edit2.Text) - single(Edit3.Text) ;
end;
```

CONVERTING FLOATING POINT TO STRING FORMAT

Of course, it would be nice to display that difference. At this point a pet peeve in all the newer languages was raised. Converting data types. The following code extract shows the problem.

```
public
  { public declarations }
  ...
  difflong : single;
  difflongstring: text;
end;
...
procedure TForm1.Button1Click(Sender: TObject);
begin
  { for no good reason, change another button's caption }
  Button2.Caption := 'CLOSE';
  { get the numeric values of the longitude and legal meridian }
  difflong := single(Edit2.Text) - single(Edit3.Text) ;
  { get the difference as text also }
  difflongstring := Text(difflong);
end;
```

```
hdiavl1.pas(55,21) Error: Incompatible types: got "TTranslateString" expected "Text"
hdiavl1.pas(55,25) Fatal: Syntax error, ";" expected but "(" found
```

Just how is numeric (single precision) data converted back into simple text. The answer is to use functions from the "sysutils" unit of Pascal. Some functions include:-

Name	Description
BCDToInt	(1357) Convert BCD number to integer
CompareMem	(1359) Compare two memory regions
FloatToStrF	(1392) Convert float to formatted string
FloatToStr	(1391) Convert float to string
FloatToText	(1394) Convert float to string
FormatFloat	(1403) Format a floating point value
GetDirs	(1406) Split string in list of directories
IntToHex	(1412) return hexadecimal representation of integer
IntToStr	(1412) return decimal representation of integer
StrToIntDef	(1442) Convert string to integer with default value
StrToInt	(1440) Convert string to integer
StrToFloat	(1439) Convert string to float
TextToFloat	(1445) Convert null-terminated string to float

The following code works well.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  { for no good reason, change another button's caption }
  Button2.Caption := 'CLOSE';

  { get the numeric values of the longitude and legal meridian }
  { single(Edit2.Text) or StrToFloat(Edit2.Text) will do conversion }
  difflong := StrToFloat(Edit2.Text) - StrToFloat(Edit3.Text) ;

  { get the difference as text also }
  StaticText4.Caption := 'standby';
  difflongstring := FloatToStr (difflong) ;
  { and drop the low order digits which are useless }
  StaticText4.Caption := LeftStr (difflongstring,3);
end;
```

And the above shows first "StrToFloat" used twice in the difference calculation, and "FloatToStr" to convert the floating point number to a string, and "LeftStr" to make the numbers look pretty.

NOTE: These functions are defined in a book that comes with the Free Pascal program used in the PASCAL files of Illustrating Shadows. The book is called:-

rtl.pdf

and is at this website:

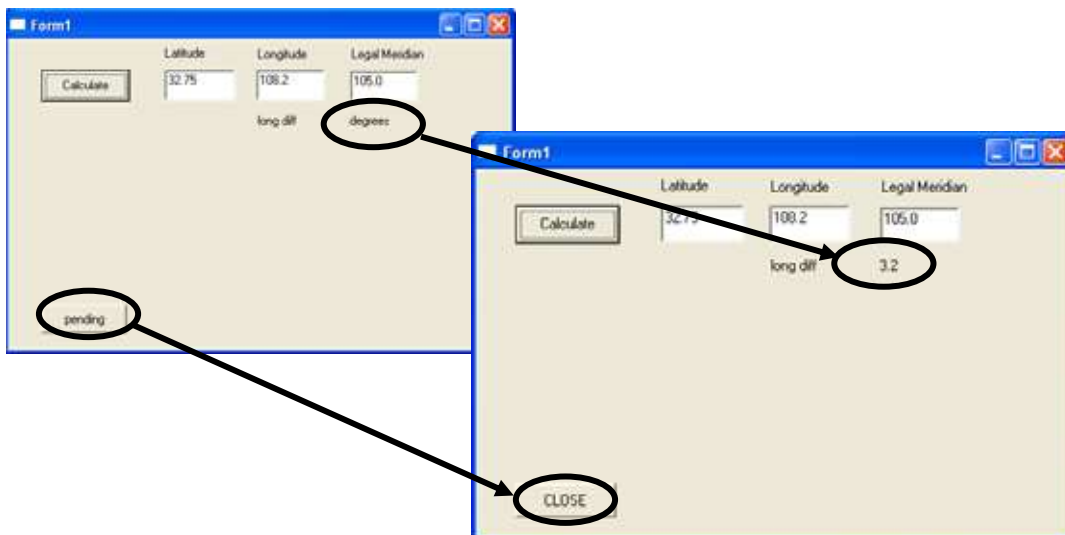
www.freepascal.org

and it lists what "uses" are needed for what functions, e.g. "uses math" for "tan, arccos", etc.

Also, some examples of Pascal are found at:-

<http://www.taoyue.com/tutorials/pascal/contents.html>

The results of the above code are shown in the before and after pictures of the FORM below.



The next problem to solve is displaying data in an array. And after that, to display the data as a graphical depiction.

DISPLAYING OUTPUT IN A TABULAR FORM AND THUS USING A LOOP

Once in STANDARD, select the next tab which is OPTIONAL. Then select TSTRINGGRID. This builds an array and you can alter the number of columns and rows. Trial and error is another pet peeve I have for the newer language implementations, and no where does the poor documentation tell you that integers for loops cannot be public global variables, and that they must be defined in the section of code in which the FOR loop is employed. That took me a couple of hours to figure out. Figuring out how to manage the cells of the array was another mammoth task, again, poor documentation.

```
. . .
. . .
. . .

{ TForm1 }

procedure TForm1.Button1Click(Sender: TObject);
var { must declare FOR variables in the section they are used in }
    irow      : integer;
    icol      : integer;
    sinlat    : single ;
    tanlha    : single ;
    hlat      : single ;
    hla       : single ;
    ii        : single ;
    jj        : single ;
    degtorad  : single ;
    radtodeg  : single ;
begin

    { for no good reason, change another button's caption }
    Button2.Caption := 'CLOSE';

    { get the numeric values of the longitude and legal meridian }
    difflong      := StrToFloat(Edit2.Text) - StrToFloat(Edit3.Text) ;

    { get the difference as text also }
    StaticText4.Caption := 'standby';
    difflongstring    := FloatToStr (difflong) ;

    { and drop the low order digits which are useless }
    StaticText4.Caption := LeftStr (difflongstring,3);

    { StringGrid1 is a TStringGrid array with 3 columns and 8 rows }

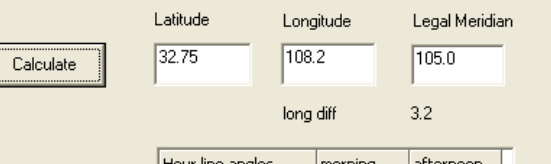
    degtorad := (2 * 3.14162) / 360 ;
    radtodeg := 360 / (2 * 3.14162) ;

    for irow := 1 to 7 do
    begin
        for icol := 1 to 2 do
        begin
            { get sin(lat) }
            sinlat := sin( degtorad * StrToFloat(Edit1.Text) ) ;

            { get the hour angle of the sun }
            ii := irow - 1 ;
```

LAZARUS programming and systems ~ an open source Delphi equivalent

The results are shown to the right.



Form1

Latitude Longitude Legal Meridian

32.75 108.2 105.0

long diff 3.2

Hour line angles	morning	afternoon
noon	1.73	1.73
11am and 1pm	10.0	-6.4
10 am and 2pm	19.4	-15.
9 am and 3 pm	31.1	-25.
8 am and 4 pm	46.9	-39.
7 am and 5 pm	68.8	-58.
6 am and 6pm	-84.	-84.

Calculate

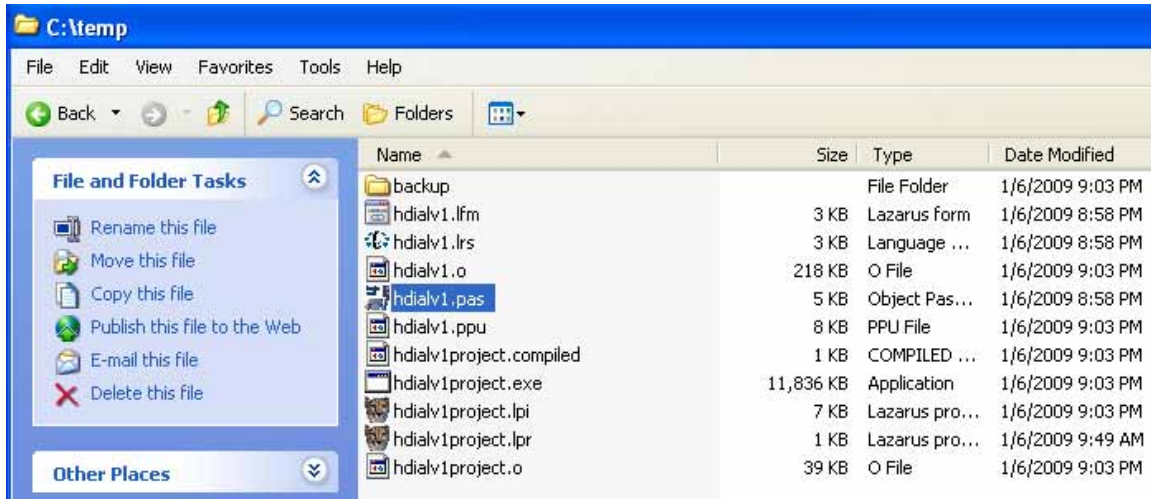
CLOSE

LAZARUS programming and systems ~ an open source Delphi equivalent

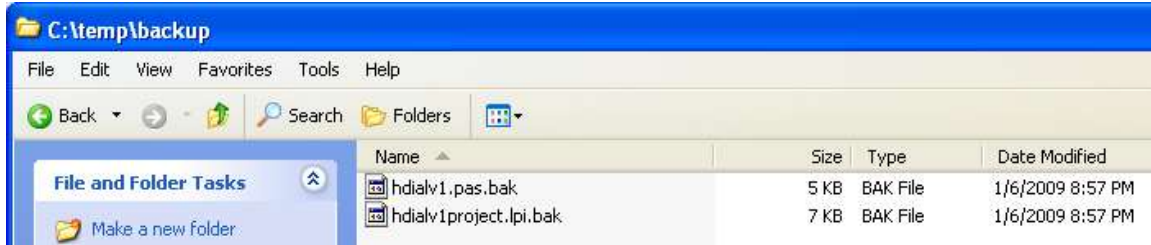
The above program is called:- `hdialv1.pas`

And was in a folder called: `c:\temp`

And the contents of that folder after all the above work was:-



And the backup folder looked like:-



This completes the LAZARUS horizontal dial for any latitude and longitude difference, in text form.

NOTE: The complete Lazarus program package is, as seen above, in one folder with a backup folder. remember, no blanks in any path names. However, unlike some of the newer programming languages, you can restore this complete folder system to any folder, it does not have to be the original folder name, and Lazarus can open it in its new folder.

NOTE: Lazarus gets upset at times and loses paths to Free Pascal Source, and loses panels such as the OBJECT INSPECTOR.

LAZARUS programming and systems ~ an open source Delphi equivalent

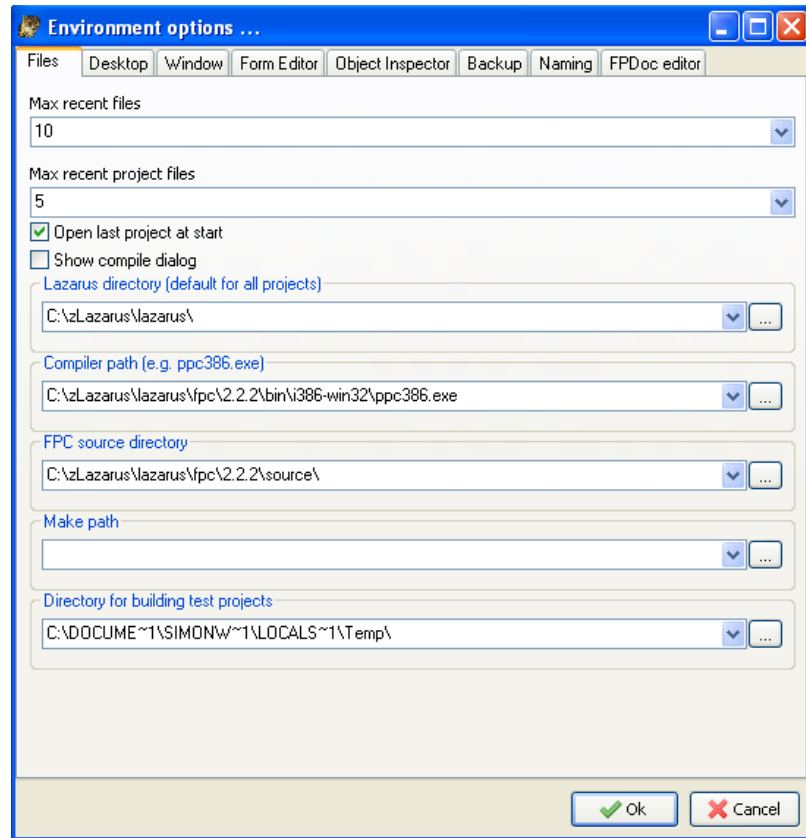
As to losing panels, VIEW, OBJECT INSPECTOR or F11 will bring that back. I have not figured how to configure it to keep it as a permanent fixture.

As to the FPC SOURCE DIRECTORY, it may help to write down the path or capture them with a screen capture as shown to the right.

If the worst comes to the worst then UNINSTALL LAZARUS and then delete the directories it used, and then completely start over from scratch, and re-install.

Lazarus has a habit of remembering where some pathc were, and not others.

There are many mysteries in the universe, and Lazarus is one of them.



Sometimes a panel will appear to vanish, such as a FORM. And VIEW, FORM does not appear to bring it back. It is still there, shrink the editing window and it will reappear.

Sometimes when executing after compiling and linking, no FORM appears so you may think that the system went to sleep. Look at the Windows bar at the bottom of the screen, chances are that it is there, so click it, and it will reappear.

One good thing about Lazarus is that it works on Windows XP SP1 as well as later service packs.

NOTE: several programs for sundials are written for Lazarus. The Horizontal and Vertical (non declining) dial program uses sin/cos for tan because it chose (for no good reason) not to include "math" in the "uses". Whereas the vertical declining dial program did include "math" in the uses, so it can use tan, arcsin, arccos and so on. A second vertical declining dial program also used graphics.

NOTE: the IBM 1401 and the IBM 360/30 assemblers, linkers, and simulators were written in Lazarus and are free Open Source on www.illustratingshadows.com and their code is well worth studying.

LAZARUS – REDUCE EXECUTABLE FILE SIZE

Lazarus files are very large, some 12mb for a small program. This is because of enormous amounts of debug data. And while the compiler can remove that, there are bugs, so that is not an option.

The solution is to locate two programs, STRIP and UPX.

STRIP is Pascal specific and locates and removes the debug data after the fact, making the executable some 20% of its original size, 12mb becomes less than 2mb.

UPX works on any executable, and shrinks the executable even further, some 2mb becomes about 0.5mb.

```
c:\whereever\lazarus\fpc\2.2.2\bin\i386-win32\strip
```

move to your folder

go to that folder

```
do "strip --strip-all system360project.exe"
```

which reduces it from 12mb to just under 2mb

```
c:\whereever\lazarus\fpc\2.2.2\bin\i386-win32\upx
```

move to your folder

go to that folder

```
do "upx system360project.exe"
```

which reduces it from just under 2mb to just under 0.5mb

WINDOWS VISTA WIN64 ISSUES

The Lazarus 32 bit system works on Windows XP as well as Vista win64, and generated code can be compressed with STRIP and UPX. However, the 64 bit version of Lazarus produces code that will not run on win32 nor on Windows XP, and cannot be compressed either with STRIP and UPX since they do not support 64 bit executables. At best, WINZIP will do a fair job of compression.

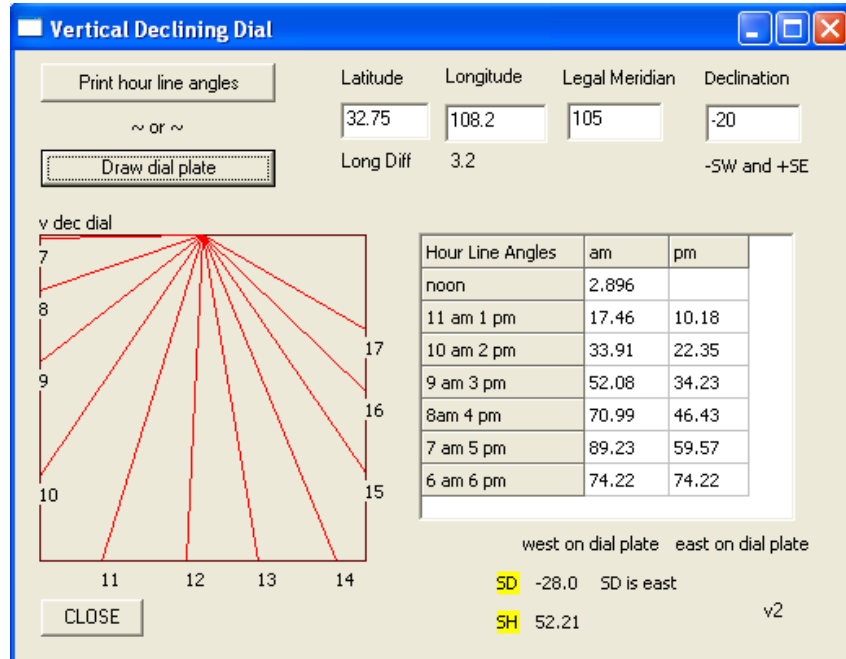
LAZARUS APEARS TO HANG AFTER “RUN”, AND THE PROGRAM DOES NOT APPEAR

Lazarus 32 bit in Vista sometimes does not bring up the program after a RUN. Click RESET DEBUGGER, when all proceeds normally. However you will get an OOPS message from Lazarus when your program ends, which you can ignore.

GRAPHICAL DISPLAY

There are two versions of the vertical decliner dial, one is tabular, the other offers both a tabular as well as a graphical depiction.

Graphics was done with a MOVE and LINE process, which had its challenges.



The problem with writing this program was that the graphics system uses integer pixel coordinated, but the TAN function returns floating point. While there is an INTEGER function, it does what it says, it returns the integer portion, but sadly it does not convert the type attribute. There is no easy way to convert floating point to integer. The technique used was to get the integer portion, convert the floating point which now has no declimal point data, to a string. And convert the string to an integer. This highlights one of the problems with the "newer" languages, namely type conversion.

The following code extract does the noon line, similar code does the actual hour lines. The noon line was chosen because of less extraneuos code.

```
{ get the hour angle of the sun for noon }
jj := difflong ;

{ get the resulting hour line angle }
hla := radtodeg*(arctan((coslat/(cosdec*1/tan(degtorad*jj))+sindec*sinlat))));

{ make numbers be positive }
if hla < 0 then hla := -hla;

{ As signs removed, noon is west of N:S if dial is west of meridian, east if east. }

{ move pen to top and center }
Canvas.Pen.Width := 1;
Canvas.Pen.Color := 255;
Canvas.Pen.Color := 250;
Canvas.MoveTo ( xbias+xyscl*halfxysiz,ybias+xyscl*0 );
```

Since this is the noon line, the Y value of the noon line endpoint is fixed as the vertical extent of the box holding the dial plate.

CONVERTING FLOATING POINT TO INTEGER (TYPE CONVERSION FRUSTRATION)

The X value is calculated using the TAN which means floating point.

```
y := 100;                                { x,y are floating point }
x := int (y * (tan(degtorad*hla))) ;      { integer part, but still float }
```

At this point, X the integer part of X, but it is still of the floating point type. And Y is floating point, and that is what creates the problem.

```
s := FloatToStrF (y,ffFixed,3,0);
```

The line above takes the Y value and converts it to a string and as it never was floating point, there is no problem.

```
yl := StrToInt (s);
```

The line above takes the string and converts it to an integer.

```
s := FloatToStrF (x,ffFixed,3,0);
xl := StrToInt (s);
```

The two lines above take X which from the code at the top of this page has had the decimal data dropped, saves the resulting integer part to a string, and then converts the string to a true integer.

```
{ draw the line }
if difflong >=0 then
begin
  xl := halfxysiz - xl;
end ;
if difflong <0 then
begin
  xl := halfxysiz + xl;
end ;
Canvas.LineTo( xbias+xyscl*xl, ybias+xyscl*yl );
```

The code above derives the correct line end point based on whether the noon line is east or west of the vertical, and then draws the line.

```
{ having drawn the line, reset to top and center }
Canvas.MoveTo ( xbias+xyscl*halfxysiz, ybias+xyscl*0 );
```

And then the pen position is reset to the dial center.

The graphics system needs a little understanding:-

KEY POINTS TO REMEMBER: 0,0 is top left
 y increases positively down

DOCUMENTATION:

<http://delphi.about.com/library/bluc/text/uc052102a.htm> [is page 1]
<http://delphi.about.com/library/bluc/text/uc052102b.htm> [is page 2]

The vertical decliner graphics program does either tabular data, or graphical depictions. The graphical display does not fill in the tables, you have to ask for that. There was no reason for that except it simplified code development.

ALTERNATIVE METHODS TO CONVERT FLOAT TO INTEGER

The following method was suggested by both a Lazarus user, as well as by one of the Lazarus developers.

Don't know if you already met this function but there is a way to convert FLOAT to INTEGER in one step...

functions TRUNC and ROUND get as parameter FLOAT and Return an INT.

Difference between them is that TRUNC just truncates the float part and leaves the integer part, while ROUND looks at the first 0 . X and rounds the number by standard ROUNDING RULE in MATH.

<http://lazarus-ccr.sourceforge.net/docs/rtl/system/trunc.html>

<http://lazarus-ccr.sourceforge.net/docs/rtl/system/round.html>

This is helpful, I would re-emphasize the comment that a practical user's guide would clearly document the common conversion functions and methods.

OTHER IMPORTANT NOTES ON WAYS OF DOING THINGS IN LAZARUS

FILE HANDLING

{ open file - if input do assign then reset } { <http://wiki.lazarus.freepascal.org/Files> }

```
assign      (infilex, 'Sim360C.bal');  
reset      (infilex);
```

{ open file - if output do assign then rewrite }

```
fn1      :=      'test.prt' ;  
assign    (outfile,'system360prt.prt');  
rewrite   (outfile);
```

Reading from a file instead of the console (keyboard) can be done by:

```
read (file_variable, argument_list);                      write (file_variable, argument_list);
```

Similarly with readln and writeln.

file_variable is declared as follows:

```
var  
...  
filein, fileout : text;
```

The text data type indicates that the file is just plain text.

After declaring a variable for the file, and before reading from or writing to it, we need to associate the variable with the filename on the disk and open the file. This can be done in one of two ways. Typically:

```
reset (file_variable, 'filename.extension');              rewrite (file_variable, 'filename.extension');
```

reset opens a file for reading, and rewrite opens a file for writing. A file opened with reset can only be used with read and readln. A file opened with rewrite can only be used with write and writeln.

Turbo Pascal introduced the assign notation. First you assign a filename to a variable, then you call reset or rewrite using only the variable.

```
assign (file_variable, 'filename.extension');  
reset (file_variable)
```

After you're done with the file, you can close it with:

```
close (File_Identifier);
```

Here's an example of a program that uses files. This program was written for Turbo Pascal and DOS, and will create file2.txt with the first character from file1.txt:

```
1: program CopyOneByteFile;
2:
3: var
4:   mychar : char;
5:   filein, fileout : text;
6:
7: begin
8:   assign (filein, 'c:\file1.txt');
9:   reset (filein);
10:  assign (fileout, 'c:\file2.txt');
11:  rewrite (fileout);
12:  read (filein, mychar);
13:  write (fileout, mychar);
14:  close(filein);
15:  close(fileout);
16: end.
```

Retrieved from <http://wiki.lazarus.freepascal.org/Files>

NOTE: ASSIGN does not work in a method invoked by a button click. It does work in a procedure, programmed before event driven code, that is called by event driven code. What follows is an **extract** from a program that works, the point to study is where file open/close is compared to where file reads and writes are located. Much has been omitted that does not help with that point.

```
unit system360code;
{$mode objfpc}{$H+}
interface

{ ***
  *** USES causes various header prototypes so functions will compile ok
  *** }

uses
  Classes, SysUtils, FileUtil, LResources, Forms, Controls, Graphics, Dialogs,
  StdCtrls, Buttons;

{ ***
  *** TYPE declares types and class models but not actual data
  *** }

type
  { TForm1 }
  TForm1 = class(TForm)
    ipl      : TButton;
    pwroff   : TButton;
  . . .
    procedure Label1Click(Sender: TObject);
  . . .
  private
    { private declarations }
  public
    { public declarations }
  end;
```

LAZARUS programming and systems ~ an open source Delphi equivalent

```
{ *****
*** VARIABLES
***      const          comes before variables          constants
***      var            comes after constants           global variables
*** ***** }

Const { fixed constants }
      maxNoSymbols = 500 ;
      maxCoreBytes = 32000 ;

Var   { variables }
      Form1: TForm1;
. . .
{ variables everyone uses }
I,J,K      :      longint;
inputFile  :      text;
outputFile :      text;
. . .
{ *****
*** PROGRAM CODE
***      subroutines  comes before event (button) driven code
***                  this is where common subroutines live
***      methods      comes after common subroutines
***                  this is where the form.button event driven code is
*** ***** }

implementation

{ *****
*** subroutines called by subsequent event driven code
***      This code has no access to the FORM and its objects
***      This code is where file i/o is done else compiler objects
*** ***** }
. . .
procedure openOutfile;
begin
      assign (outputFile, 'system360.lst');
      rewrite(outputFile);
      { write (outputFile, mychar);      *** can be done is event driven code *** }
end;
. . .
procedure closeOutfile;
begin
      close (outputFile);
end;

{ *****
*** methods are event driven by buttons and that code in turn may call
*** some of the code in the previous area
***      this code does have access to the FORM and its objects
*** ***** }
{ TForm1 }

{ ***
*** POWER ON
***
}
procedure TForm1.pwronClick(Sender: TObject);
begin
      pwroff.visible      := true ;
      ipl.visible         := true ;
end;
```

LAZARUS programming and systems ~ an open source Delphi equivalent

```
{ ***
  *** IPL LOAD - determine to 1 assemble a file
  ***
}
procedure TForm1.iplClick(Sender: TObject);
begin
  . . .
  { open the input file - cant do ASSIGN here as compiler has issues }
  openOutfile();
  write (outputFile, 'test');
  closeOutfile();

  { this end is the end of ipl's begin }
end;

...

procedure TForm1.Edit1Change(Sender: TObject);
begin
end;
procedure TForm1.Label1Click(Sender: TObject);
begin
end;
procedure TForm1.Label3Click(Sender: TObject);
begin
end;

initialization
  {$I system360code.lrs}

end.
```

ARRAYS WITH MORE THAN ONE COLUMN PER ROW

```
{ ***
  *** S Y M B O L    T A B L E    F O R    A S S E M B L E R
  *** }
{ Assembler phase: symbol table: can do it two ways

  one way:  one table for symbols, the other for the address, the
             common link is the index, so symbol and name are not
             contiguous.
  another:  one table whose entries are a record, and the record
             has two fields, one the symbol, the other the address
  both work
}
```

```
{ symbol table for labels in source code we will assemble, and core
location }
```

```
{ first method is }
```

```
symTableName    : ARRAY [1..maxNoSymbols] OF string;
symTableAddr    : ARRAY [1..maxNoSymbols] OF Longint;
```

```
{ second method is one array of a record type... }
```

```
symTab          : ARRAY [1..maxNoSymbols] OF
                  RECORD
                    Name : ARRAY [0..8] OF CHAR;
                    Addr : integer
                  END;
```


And table access can be:-

```
{ ***
  *** CLEAR SYMBOL TABLE FOR ASSEMBLER
  *** }

{ Assembler phase: symbol table: can do it two ways
  one way: one table for symbols, the other for the address, the
            common link is the index, so symbol and name are not
            contiguous.
  another: one table whose entries are a record, and the record
            has two fields, one the symbol, the other the address
  both work
}

{ first method is }

for I := 0 TO maxNoSymbols DO
begin
    symTableName [I] := '          ';
    symTableAddr [i] := 0 ;
end ;

(* NOW THE BLANKS IN THE MNEMONIC FIELDS ARE REPLACED WITH THE *)
(* CHARACTER WHOSE CODE IS LOWC. IN THIS WAY, WE ASSURE *)
(* THAT STRING COMPARISONS WILL WORK CORRECTLY REGARDLESS OF *)
(* PARTICULAR MACHINE IN USE. *)

FOR I := 1 TO maxNoSymbols DO
    FOR J := 0 TO 8 DO
        IF symTableName[J] = ' '
            THEN symTableName[J] := chr(0) ;

{ second method is one array of a record type... }

{ we chose one array of a record type... }
for I := 1 TO maxNoSymbols DO
begin
    symTab[i].Name      := '          '; { blank name }
    symTab[i].Name[8]   := ' ';
    symTab[i].Addr      := 0;             { address of symbol }
end;
(* Make blanks 0 so string compares good on all systems *)
FOR I := 1 TO maxNoSymbols DO
    FOR J := 0 TO 8 DO
        IF symTab[I].Name[J] = ' '
            THEN symTab[I].Name[J] := chr(0) ;
```